Programming Languages

2nd edition

Tucker and Noonan

Chapter 3
Lexical and Syntactic Analysis

Syntactic sugar causes cancer of the semicolon.
A. Perlis

Contents

Syntactic Analysis

Phase also known as: parser

Purpose is to recognize source structure

Input: tokens

Output: parse tree or abstract syntax tree

A recursive descent parser is one in which each nonterminal in the grammar is converted to a function which recognizes input derivable from the nonterminal.

Program Structure consists of:

Expressions: `x + 2 * y`

Assignment Statement: `z = x + 2 * y`

Loop Statements:

```
while (i < n) a[i++] = 0;
```

Function definitions

Declarations: `int i;`

Assignment → *Identifier* = *Expression*

Expression → *Term* { *AddOp* *Term* }

AddOp → + | -

Term → *Factor* { *MulOp Factor* }

MulOp → * | /

Factor → [*UnaryOp*] *Primary*

UnaryOp → - | !

Primary → *Identifier* | *Literal* | (*Expression*)

First Set

Augmented form: S – start symbol

$S' \rightarrow S \ \$$

First(X) is set of leftmost terminal symbols derivable from X

$First(X) = \{ a \in T \mid X \rightarrow^* a\, w,\ w \in (N \cup T)^* \}$

Since there are no production rules for terminal symbols:

$First(a) = a,\ a \in T$

For w = X1 ... Xn V ...

First(w) = First(X1) ∪ ... ∪ First(Xn) ∪ First(V)

where X1, ..., Xn are nullable

and V is not nullable

A is *nullable* if it derives the empty string.

Nullable Algorithm

```
Set nullable = new Set( );
do { oldSize = nullable.size( );
     for (Production p : grammar.productions( ))
          if (p.length( ) == 0)
               nullable.add(p.nonterminal( ));
          else
               << check righthand side >>
} while (nullable.size( ) > oldSize);
```

Check Righthand Side

```
{
    boolean  allNull = true;
    for (Symbol t : p.rule( ))
        if (! nullable.contains(t))
            allNull = false;
    if (allNull)
        nullable.add(p.nonterminal( ));
}
```

Rewrite Grammar

Augmented form

Abbreviate symbols

Replace meta constructs with nonterminals

Transformed Grammar

S → A $

A → i = E ;

E → T E'

E' → | AO T E'

AO → + | -

T → F T'

T' → MO F T'

MO → * | /

F → F' P

F' → | UO

UO → - | !

P → *i* | *l* | (E)

Compute Nullable Symbols

Pass	Nullable
1	E' T' F'
2	E' T' F'

Left Dependency Graph

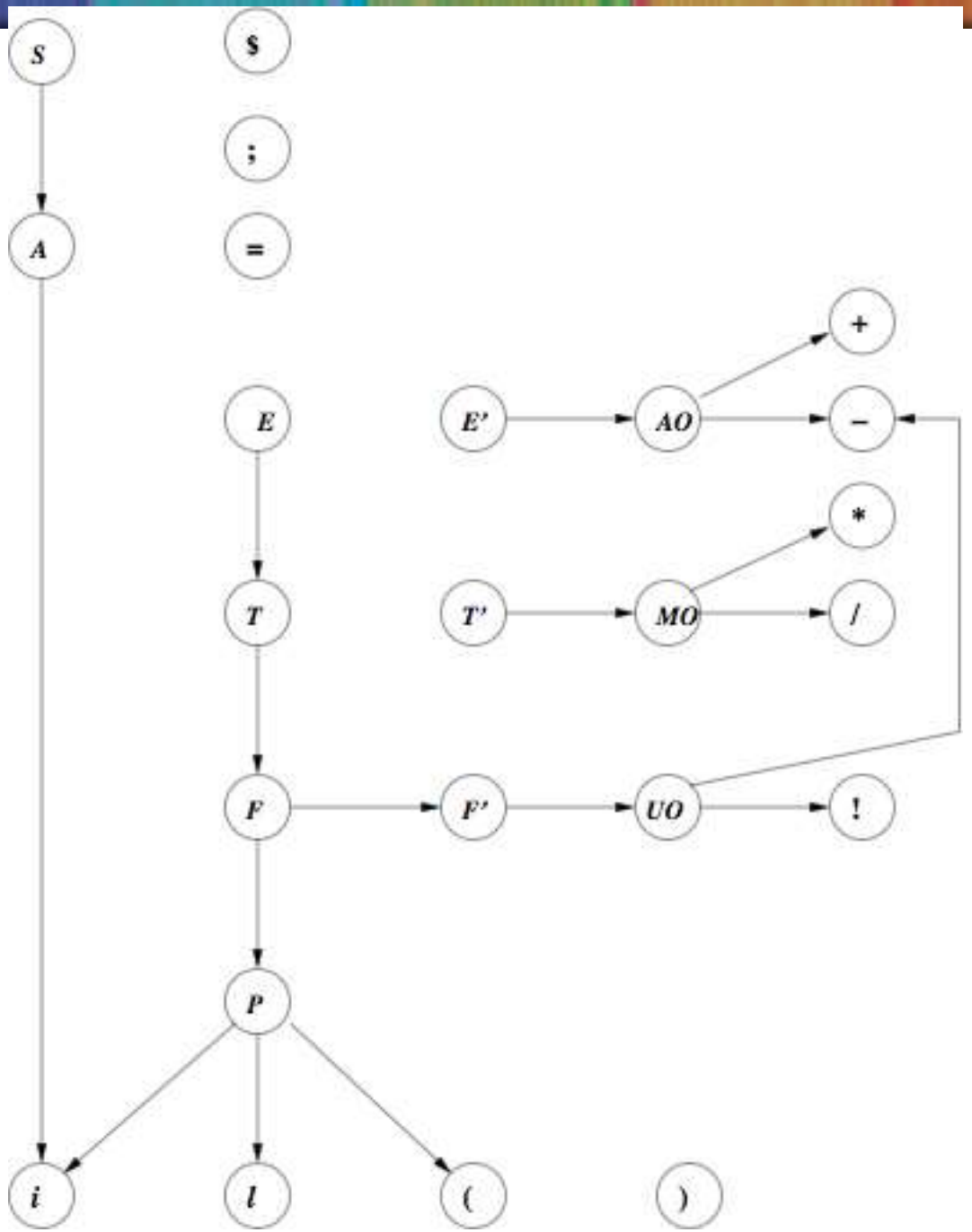
For each production:

$A \rightarrow V1 \ldots Vn\ X\ w$

V1, ... Vn nullable

draw an arc from A to X.

Note: you also get arcs from A to V1, ..., A to Vn


S
$
;
A
=
E
E'
AO
+
–
T
T'
MO
*
/
F
F'
UO
!
P
i
l
(
)

Nonterminal	First	Nonterminal	First
A	i	UO	! -
E	! - i l (	P	i l (
E’	+ -		
AO	+ -		
T	! - i l (		
T’	* /		
MO	* /		
F	! - i l (		
F’	! -		

Recursive Descent Parsing

Method/function for each nonterminal

Recognize longest sequence of tokens derivable from the nonterminal

Need an algorithm for converting productions to code

Based on EBNF

T(EBNF) = Code: A → w

1 If w is nonterminal, call it.

2 If w is terminal, match it against given token.

3 If w is { w' }:

while (token in First(w')) T(w')

4 If w is: w1 | ... | wn,

switch (token) {

case First(w1): T(w1); break;

...

case First(wn): T(wn); break;

5 Switch (cont.): If some wi is empty, use:

default: break;

Otherwise

default: error(token);

6 If w = [w'], rewrite as (| w') and use rule 4.

7 If w = X1 ... Xn, T(w) =

T(X1); ... T(Xn);

Augmentation Production

Gets first token

Calls method corresponding to original start symbol

Checks to see if final token is end token

E.g.: end of file token

```
private void match (int t) {
    if (token.type() == t)
        token = lexer.next();
    else
        error(t);
}
```

```
private void error(int tok) {
    System.err.println(
        “Syntax error: expecting”
        + tok + “; saw: ”  + token);
    System.exit(1);
}
```

```
private void assignment( ) {
   // Assignment → Identifier = Expression ;
   match(Token.Identifier);
   match(Token.Assign);
   expression( );
   match(Token.Semicolon);
}
```

```
private void expression( ) {
   // Expression → Term { AddOp Term }
   term( );
   while (isAddOp()) {
      token = lexer.next( );
      term( );
   }
}
```

Linking Syntax and Semantics

Output: parse tree is inefficient

- One nonterminal per precedence level

Shape of parse tree that is important

Discard from Parse Tree

Separator/punctuation terminal symbols

All trivial root nonterminals

Replace remaining nonterminal with leaf terminal

Abstract Syntax Example

Assignment = *Variable* target; *Expression* source

Expression = *Variable* | *Value* | *Binary* | *Unary*

Binary = *Operator* op; *Expression* term1, term2

Unary = *Operator* op; *Expression* term

Variable = *String* id

Value = *Integer* value

Operator = + | - | * | / | !

```
abstract class Expression { }

class Binary extends Expression {
    Operator op;
    Expression term1, term2;
}

class Unary extends Expression {
    Operator op; Expression term;
}
```

Modify T(A → w) to Return AST

1. Make A the return type of function, as defined by abstract syntax.
2. If w is a nonterminal, assign returned value.
3. If w is a non-punctuation terminal, capture its value.
4. Add a return statement that constructs the appropriate object.

```
private Assignment assignment( ) {
   // Assignment → Identifier = Expression ;
   Variable target = match(Token.Identifier);
   match(Token.Assign);
   Expression source = expression( );
   match(Token.Semicolon);
   return new Assignment(target, source);
}
```

```
private String match (int t) {
    String value = Token.value( );
    if (token.type( ) == t)
        token = lexer.next( );
    else
        error(t);
    return value;
}
```